SAMSUNG

if (oops) { do_not_panic(); }

Lucky Tyagi

Samsung Semiconductor India R&D Center

Agenda

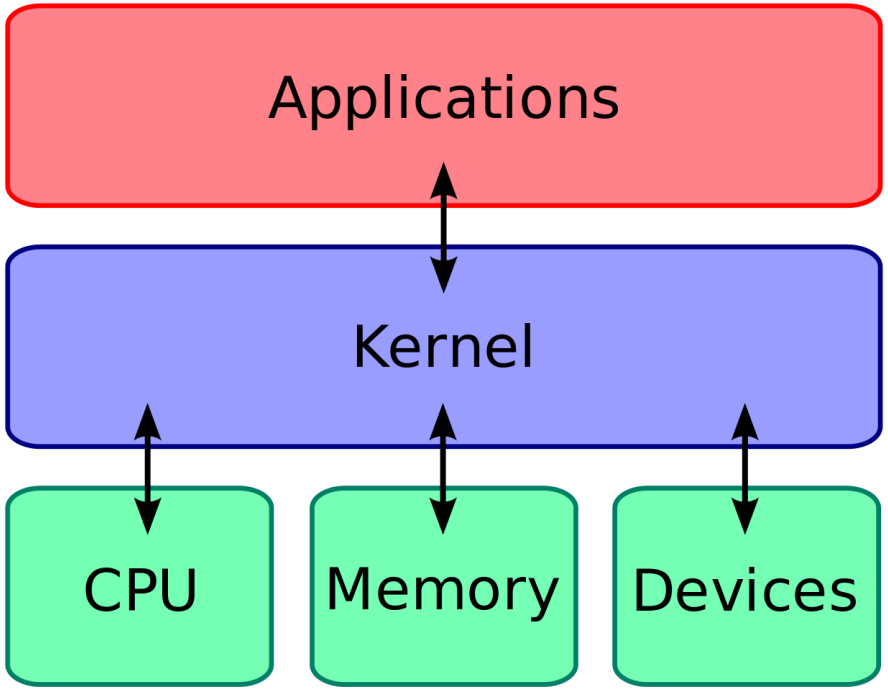
- Introduction
- Kernel Panic
- Debugging OOPS!
  - KEXEC, KDUMP
  - ADDR2LINE, OBJDUMP, GDB
  - KDB, KGDB
- Summary

Introduction

- What is Linux Kernel?
  - Free, Open-Source and Unix-like OS Kernel
  - Core interface between hardware and processes
  - Manages resources as efficiently as possible

- The kernel is so named because—like a seed inside a hard shell—it exists within the OS and controls all the major functions of the hardware, whether it's a phone, laptop, server, or any other kind of computer.

Introduction



Introduction

❑ Sub-systems of Linux Kernel

- **The Process Scheduler**: responsible for fairly distributing the CPU time
- **The Memory Management Unit**: responsible for proper distribution of the memory resources
- **The Virtual File System**: responsible for providing a unified interface to access stored data
- **The Networking Unit**: allows Linux systems to connect to other systems over a network
- **Inter-Process Communication Unit**: processes communicate with each other and with the kernel to coordinate their activities

Kernel Panic

- A safety measure taken by Kernel upon detecting an internal fatal error
- Situation when the kernel can't load properly and therefore the system fails to boot
- OOPS: A Linux Kernel problem, bad enough to affect system reliability
- Causes
  - Software bug in the OS
  - Hardware Failure
  - Malfunctioning RAM
  - Incompatible device driver
  - Corrupt RFS
  - Init Process fails or terminates

Kernel Panic

- The panic() routine
  - Handles Kernel Panic
  - Output an error message to the console
  - Dump an image of kernel memory to disk for debugging
  - Either wait for manual reboot or initiate an automatic reboot

- OOPS contains information about
  - IP which caused the fault
  - Register Status
  - Process
  - CPU number
  - Stack Trace of functions

Kernel Panic


```
static noinline void do_oops(void)
{
    *(int*)0x42 = 'a';
}

static int foo_oops_init(void)
{
    pr_info("oops_init\n");
    do_oops();

    return 0;
}
```

```
static void foo_oops_exit(void)
{
    pr_info("oops exit\n");
}

module_init(foo_oops_init);
module_exit(foo_oops_exit);
```

Kernel Panic


```
# insmod oops.ko
BUG: unable to handle kernel NULL pointer dereference at 000
00042
IP: do_oops+0x8/0x10 [oops]
*pde = 00000000
Oops: 0002 [#1] SMP
Modules linked in: oops(O+)
CPU: 0 PID: 234 Comm: insmod Tainted: G          O    4.15.0+ #3
Hardware name: QEMU Standard PC (i440FX + PIIX, 1996), BIOS
Ubuntu-1.8.2-1ubuntu1 04/01/2014
EIP: do_oops+0x8/0x10 [oops]
EFLAGS: 00000292 CPU: 0
EAX: 00000061 EBX: 00000000 ECX: c7ed3584 EDX: c7ece8dc
ESI: c716c908 EDI: c8816010 EBP: c7257df0 ESP: c7257df0
DS: 007b ES: 007b FS: 00d8 GS: 0033 SS: 0068
CR0: 80050033 CR2: 00000042 CR3: 0785f000
Call Trace:
foo_oops_init+0x17/0x20 [oops]
do_one_initcall+0x37/0x170
? cache_alloc_debugcheck_after.isra.19+0x15f/0x2f0
? __might_sleep+0x32/0x90
? trace_hardirqs_on_caller+0x11c/0x1a0
? do_init_module+0x17/0x1c2
? kmem_cache_alloc+0xa4/0x1e0
? do_init_module+0x17/0x1c2
do_init_module+0x46/0x1c2
load_module+0x1f45/0x2380
SyS_init_module+0xe5/0x100
do_int80_syscall_32+0x61/0x190
entry_INT80_32+0x2f/0x2f
EIP: 0x44902cc2
EFLAGS: 00000206 CPU: 0
EAX: ffffffda EBX: 08afb050 ECX: 0000eef4 EDX: 08afb008
ESI: 00000000 EDI: bf914dbc EBP: 00000000 ESP: bf914c1c
DS: 007b ES: 007b FS: 0000 GS: 0033 SS: 007b
Code: <a3> 42 00 00 00 5d c3 90 55 89 e5 83 ec 04 c7 04 24 24
70 81 c8 e8
EIP: do_oops+0x8/0x10 [oops] SS:ESP: 0068:c7257df0
CR2: 0000000000000042
---[ end trace 011848be72f8bb42 ]---
Killed
```

Debugging OOPS!

- PRINTK
  - Standard function for printing messages and usually the most basic way of tracing and debugging
- ADDR2LINE
  - Translates addresses into file names and line numbers
- OBJDUMP
  - Display information from OBJECT file
- GDB
  - The GNU debugger

Debugging OOPS!

ADDR2LINE

```
$ addr2line -e oops.o  0x08
$ skels/debugging/oops/oops.c:5
```

OBJDUMP

```
$ cat /proc/modules
oops 20480 1 - Loading 0xc8816000 (O+)

$ objdump -dS --adjust-vma=0xc8816000 oops.ko
c8816000:      b8 61 00 00 00          mov    $0x61,%eax

static noinline void do_oops(void)
{
c8816005:      55                      push   %ebp
c8816006:      89 e5                   mov    %esp,%ebp
*(int*)0x42 = 'a';
c8816008:      a3 42 00 00 00          mov    %eax,0x42
```

GDB

```
$ gdb ./vmlinux

(gdb) list *(do_panic+0x8)
0xc1244138 is in do_panic (lib/test_panic.c:8).
3
4 static struct timer_list panic_timer;
5
6 static void do_panic(struct timer_list *unused)
7 {
8       *(int*)0x42 = 'a';
9 }
10
11      static int so2_panic_init(void)
```

Debugging OOPS!

- KEXEC: Boot into another kernel from an existing kernel
- KDUMP: KEXEC based crash-dumping mechanism

KERNEL CONFIG	TARGET SYSTEM
CONFIG_RELOCATABLE=y	$ sudo apt update && sudo apt upgrade
CONFIG_KEXEC=y	
CONFIG_CRASH_DUMP=y	
CONFIG_DEBUG_INFO=y	$ sudo apt install gcc make binutils linux-headers-`uname -r` kdump-tools crash `uname -r`-dbg
CONFIG_MAGIC_SYSRQ=y	
CONFIG_PROC_VMCORE=y	

Debugging OOPS!

- Enable kexec to handle reboots
- Enable kdump to run and load at system boot
- Configuring kdump
  - /etc/default/kdump-tools
  - /etc/default/grub.d/kdump-tools.default
- Verify kdump environment
  - $ sudo dmesg |grep -i crash
- Dry run test
  - $ sudo kdump-config test

Debugging OOPS!

- Force a kernel panic over SysRq
  - $ echo "c" | sudo tee -a /proc/sysrq-trigger
- Kernel loaded over kexec will save
  - State of the system
  - Memory
  - CPU
  - Loaded modules and more
- System reboots automatically to a functional state

Debugging OOPS!


```
$ cd /var/crash/ ; ls
202205221645    kexec_cmd

$ sudo crash dump.202205221645 /usr/lib/debug

KERNEL: /usr/lib/debug/vmlinux-4.9.0-8-amd64
   DUMPFILE: dump.202205221645  [PARTIAL DUMP]
       CPUS: 4
       DATE: Sun May 22 16:45:21 2022
     UPTIME: 00:04:09
LOAD AVERAGE: 0.00, 0.00, 0.00
      TASKS: 100
   NODENAME: deb-panic
    RELEASE: 4.9.0-8-amd64
    VERSION: #1 SMP Debian 4.9.144-3 (2022-05-22)
    MACHINE: x86_64  (2592 Mhz)
     MEMORY: 4 GB
      PANIC: "sysrq: SysRq : Trigger a crash"
        PID: 563
    COMMAND: "tee"
       TASK: ffff88e69628c080 [THREAD_INFO: ffff88e69628c080]
        CPU: 2
      STATE: TASK_RUNNING (SYSRQ)
```

```
crash> bt
PID: 563    TASK: ffff88e69628c080  CPU: 2   COMMAND: "tee"
 #0 [ffffa67440b23ba0] machine_kexec at ffffffffa0c53f68
 #1 [ffffa67440b23bf8] __crash_kexec at ffffffffa0d086d1
 #2 [ffffa67440b23cb8] crash_kexec at ffffffffa0d08738
 #3 [ffffa67440b23cd0] oops_end at ffffffffa0c298b3
 #4 [ffffa67440b23cf0] no_context at ffffffffa0c619b1
 #5 [ffffa67440b23d50] __do_page_fault at ffffffffa0c62476
 #6 [ffffa67440b23dc0] page_fault at ffffffffa121a618
    [exception RIP: sysrq_handle_crash+18]
    RIP: ffffffffa102be62  RSP: ffffa67440b23e78  RFLAGS: 00010282
    RAX: ffffffffa102be50  RBX: 0000000000000063  RCX: 0000000000000000
    RDX: 0000000000000000  RSI: ffff88e69fd10648  RDI: 0000000000000063
```

Debugging OOPS!

- KDB: In-Kernel Debugger
- KGDB: Kernel GNU Debugger
- 2.6.26 - KGDB was merged
- 2.6.35 - KDB was merged, and uses the same backend as KGDB

- It is possible to use either of the debuggers and dynamically transition between them if you configure the kernel properly at compile and runtime

Debugging OOPS!

- KDB
  - Not a source level debugger
  - Simplistic shell-style interface which you can use on a system console
  - Inspect memory, registers, process lists, dmesg
  - Set breakpoints to stop in a certain location
  - Mainly aimed at doing some analysis to aid in development or diagnosing kernel problems

Debugging OOPS!

- KGDB
  - Source level debugger for the Linux kernel used along with gdb
  - Similar to the way an application developer would use gdb to debug an application
  - It is possible to place breakpoints in kernel code and perform some limited execution stepping
  - Two machines are required for using kgdb, a development machine and a target machine
  - The kernel to be debugged runs on the target machine
  - The development machine runs an instance of gdb against the vmlinux
  - In gdb the developer specifies the connection parameters and connects to kgdb

- Kernel config options

KGDB	KDB
# CONFIG_DEBUG_RODATA is not set	# CONFIG_DEBUG_RODATA is not set
CONFIG_FRAME_POINTER=y	CONFIG_FRAME_POINTER=y
CONFIG_KGDB=y	CONFIG_KGDB=y
CONFIG_KGDB_SERIAL_CONSOLE=y	CONFIG_KGDB_SERIAL_CONSOLE=y
	CONFIG_KGDB_KDB=y
	CONFIG_KDB_KEYBOARD=y

Debugging OOPS!

- Kernel Debugger Boot Arguments
  - kgdboc: primary mechanism to configure how to communicate from gdb to kgdb as well as the devices you want to use to interact with the kdb shell
  - kgdbwait: makes kgdb wait for a debugger connection during booting of a kernel
  - kgdbcon: allows you to see printk() messages inside gdb while gdb is connected to the kernel

Debugging OOPS!

- KGDBOC
  - Kgdboc may be configured as a kernel built-in or a kernel loadable module
  - Can only make use of kgdbwait and early debugging if you build kgdboc into the kernel as a built-in
- Usage: kgdboc=[kms][[,]kbd][[,]serial_device][,baud]
  - Kernel boot argument: kgdboc=<tty-device>,[baud]
    - kgdboc=ttyS0,115200
  - Loadable module: modprobe kgdboc kgdboc=<tty-device>,[baud]
    - echo ttyS0 > /sys/module/kgdboc/parameters/kgdboc # ENABLE
    - echo "" > /sys/module/kgdboc/parameters/kgdboc # DISABLE

Debugging OOPS!

- KGDBCON
  - KDB does not use KGDBCON feature
  - Two ways to activate
    - Kernel Boot Arguments: kgdbcon
    - Sysfs (before configuring I/O driver): echo 1 > /sys/module/kgdb/parameters/kgdb_use_con
  - Cannot use kgdboc + kgdbcon on a tty that is an active system console
    - console=ttyS0,115200 kgdboc=ttyS0 kgdbcon
  - It is possible to use this option with kgdboc on a tty that is not a system console

Using KDB

- Boot kernel with arguments:
  - console=ttyS0,115200 kgdboc=ttyS0,115200
- Or, configure kgdboc in sysfs:
  - echo ttyS0 > /sys/module/kgdboc/parameters/kgdboc
- Enter the kernel debugger manually or by waiting for an oops or fault:
  - echo g > /proc/sysrq-trigger
- From the kdb prompt, run the "help" command
  - lsmod: Shows where kernel modules are loaded
  - ps: Displays only the active processes
  - ps A: Shows all the processes
  - summary: Shows kernel version info and memory usage
  - bt: Get a backtrace of the current process using dump_stack()
  - dmesg: View the kernel syslog buffer
  - go: Continue the system

Using KGDB

- Boot kernel with arguments: kgdboc=ttyS0,115200
- Or, configure kgdboc in sysfs:
  - echo ttyS0 > /sys/module/kgdboc/parameters/kgdboc
- Enter the kernel debugger manually or by waiting for an oops or fault:
  - echo g > /proc/sysrq-trigger
- Connect from from gdb
  - using a directly connected port:
    - $ gdb ./vmlinux
    - (gdb) set remotebaud 115200
    - (gdb) target remote /dev/ttyS0
  - On TCP Port 2012:
    - $ gdb ./vmlinux
    - (gdb) target remote 192.168.2.2:2012

Using KGDB

- Once connected, you can debug a kernel the way you would debug an application program

- Enable gdb to be verbose about its target communications
  - (gdb) set debug remote 1

Interoperability

- Switching from KGDB to KDB
  - (gdb) $3#33
  - (gdb) maintenance packet 3
- Switching from KDB to KGDB
  - From KDB issue the command: kgdb

Summary

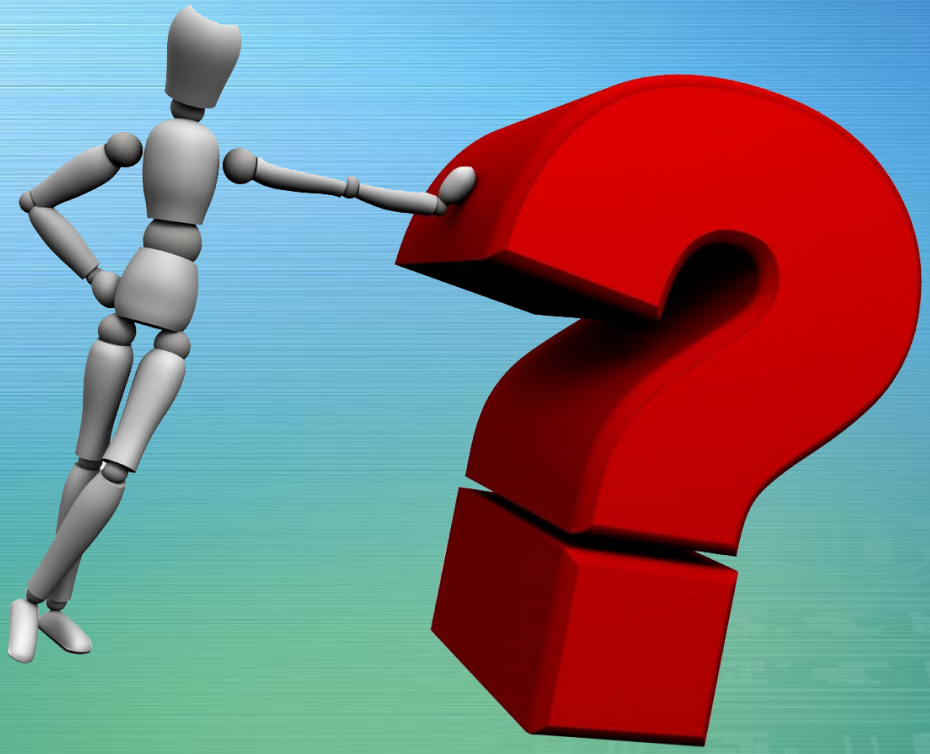
This talk is intended for developers who have just begun their journey in Linux Kernel Development.

A general methodology of debugging a kernel panic is discussed here. After triggering a simple soft panic, a standard approach is followed explaining the various debugging tools and their usage to root-cause the issue.

I remarked to Dennis that easily half the code I was writing in Multics was error recovery code.
He said, "We left all that stuff out. If there's an error, we have this routine called panic, and when it is called, the machine crashes, and you holler down the hall, 'Hey, reboot it.'"

Tom van Vleck

Any Questions ?



THANK YOU